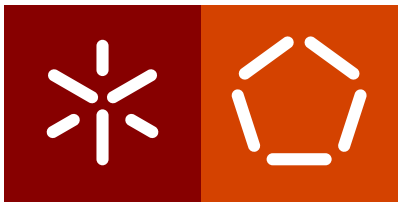
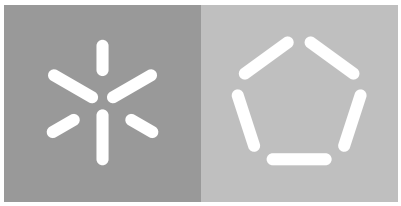


Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Carlos Vieira Morais

Recuperação de transações em bases de dados NoSQL

Junho de 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Carlos Vieira Morais

Recuperação de transações em bases de dados NoSQL

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação efetuada sob a orientação de
José Orlando Pereira
Ana Nunes Alonso

Junho de 2018

AGRADECIMENTOS

Em primeiro lugar queria agradecer ao meu orientador, o professor José Orlando Pereira e à minha supervisora Ana Nunes Alonso por me terem aceite nesta dissertação, pela ajuda, tempo despendido e pelas discussões sempre produtivas sobre o trabalho feito.

A todos os amigos que tive oportunidade de fazer durante o meu percurso académico, em especial ao José Pereira que me acompanhou durante estes 6 anos. Agradeço também aos meus amigos de Guimarães pela motivação dada e estarem sempre comigo ao longo destes anos.

Por último, queria agradecer à minha família, em especial aos meus pais que me deram a oportunidade de chegar até aqui.

RESUMO

Com o amadurecimento e ampla utilização das bases de dados NoSQL tem havido um interesse crescente na adição de transações multi-linha, que proporcionem as propriedades ACID sem comprometer o desempenho e capacidade de escala destes sistemas. Apesar das propostas nesta área assentarem em técnicas bem conhecidas de bases de dados, como a multi-versão e a recuperação, a sua aplicação está agora enquadrada em pressupostos diferentes, não sendo claro que os compromissos tradicionais se mantenham.

Neste contexto, este trabalho sistematiza os compromissos relacionados com a escolha de um mecanismo de recuperação, que garante que as alterações efetuadas por uma transação confirmada persistem atomicamente. Além de analisar qual o impacto na arquitetura do sistema da escolha do mecanismo de recuperação, comparamos experimentalmente as alternativas mais interessantes com diferentes cargas de trabalho.

ABSTRACT

With the maturing and extensive use of NoSQL databases there has been a growing interest in adding multi-line transactions that provide ACID properties without compromising the performance and scalability of these systems. Although proposals in this area are based on traditional database techniques such as multi-version and recovery, their application is now framed in different assumptions, and it is not clear that traditional compromises remain.

In this context, this work systematizes the compromises related to the choice of a recovery mechanism, which guarantees that the changes made by a commited transaction persist atomically. In addition to analyzing the impact on the system architecture of the choice of the recovery mechanism, we compared experimentally the most interesting alternatives with different workloads.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE ALGORITMOS

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

O crescimento da Internet e o surgimento de novas tecnologias web transformaram nos últimos anos o paradigma computacional. Uma nova geração de aplicações web, enfrenta o desafio de servir milhões de utilizadores distribuídos pelo mundo e ao mesmo tempo disponibilizar um serviço sempre disponível e confiável. Com o elevado número de utilizadores, aumentou também o volume de dados, com novas exigências para o seu armazenamento e processamento.

Os sistemas de gestão de bases de dados relacionais fornecem poderosos mecanismos para armazenar e consultar dados estruturados com garantias transacionais. Contudo, não foram desenvolvidos a pensar neste enorme volume de informação e falham também no suporte a elevada escalabilidade e disponibilidade, uma vez que se baseiam numa arquitetura centralizada e rígida.

Surgiram assim sistemas de bases de dados não relacionais, conhecidos como NoSQL. Estes sistemas foram desenvolvidos para serem distribuídos, escaláveis e altamente disponíveis. Contudo o aumento de desempenho destes sistemas foi conseguido ao relaxar as garantias transacionais e/ou restringindo significativamente o âmbito possível de cada transação.

Embora ofereçam uma coerência inevitável dos dados, sistemas NoSQL têm obtido grande sucesso, uma vez que várias aplicações não requerem uma coerência forte dos dados. Alcançar a escalabilidade dos sistemas NoSQL e garantir a coerência dos dados e garantias transacionais dos RDBMS tem sido o alvo de investigação ao longo dos últimos anos.

1.2 MOTIVAÇÃO

Recentemente, surgiram várias propostas de implementação de transações ACID em bases de dados não relacionais. Estas propostas seguem técnicas bem conhecidas de bases de dados, como a multi-versão e a recuperação, sendo que a sua aplicação está agora enquadrada em pressupostos diferentes, não sendo claro que os compromissos tradicionais se mantenham.

1.3 OBJETIVOS

Neste trabalho pretendemos comparar objetivamente os compromissos relacionados com a escolha do mecanismo de recuperação num sistema de base de dados NoSQL, analisar qual o impacto na arquitetura do sistema do mecanismo de recuperação e comparar as diferentes alternativas com diferentes cargas de trabalho com a utilização de um *benchmark*.

1.4 ESTRUTURA DA DISSERTAÇÃO

Este documento está organizado em 5 capítulos. No capítulo 2 é feita uma análise ao estado da arte relativamente às transações, bases de dados NoSQL distribuídas e transações em bases de dados NoSQL. No capítulo 3 é apresentada a arquitetura e implementação dos protótipos para as alternativas *undo* e *redo*, no final é feita uma discussão ao impacto de ambas as alternativas na arquitetura do sistema. No capitulo 4 é feita uma avaliação e comparação às alternativas. No capítulo 5 concluímos o trabalho.

2

ESTADO DA ARTE

2.1 TRANSAÇÕES

Uma transação é uma sequência de operações de leitura e escrita executada como uma única unidade de trabalho. Depois de iniciada e executadas as operações que a compõem, é feito o pedido de confirmação da transação ao gestor de transações, responsável por determinar o resultado. Uma transação que executa corretamente e produz efeito, diz-se confirmada, caso contrário diz-se abortada. Uma transação que tenha iniciado a sua execução mas ainda não tenha sido confirmada ou abortada diz-se ativa. Duas transações T_1 e T_2, em que T_1 tenha iniciado antes de T_2 mas terminado depois do início de T_2 dizem-se concorrentes.

Por definição uma transação deve ter quatro propriedades fundamentais, conhecidas como ACID [17], de forma a providenciar a confiabilidade dos dados. Estas propriedades definem que as transações são protegidas de problemas de concorrência e falhas de *software* e *hardware*. As propriedades referidas no acrónimo ACID são:

ATOMICIDADE A transação deve ser uma unidade atómica de trabalho, isto é, ou todas as operações da transação são executadas ou nenhuma é.

COERÊNCIA Quando concluída, a transação deve deixar o sistema num estado coerente.

ISOLAMENTO A execução de uma transação deve ser isolada da execução de outras transações concorrentes, ou seja, nenhuma transação deve interferir com outras transações concorrentes e as alterações de uma transação incompleta não devem ser visíveis a outras transações.

DURABILIDADE Os efeitos de uma transação executada corretamente devem de permanecer no sistema, mesmo na presença de falhas no sistema.

2.1.1 *Arquitetura de um SGBD tradicional*

A arquitetura de um SGBD transacional tradicional [16] assume que a base de dados está armazenada em disco. Esta informação está organizada fisicamente em blocos de tamanho

fixo, que podem ser lidos e escritos no contexto de transações. Durante a execução de uma interrogação, os blocos que contêm a informação necessária são lidos para memória principal onde são consultados e possivelmente modificados. Como a memória principal é limitada, os blocos em memória são inevitavelmente removidos para dar lugar a outros. No caso de terem sido modificados, são escritos de volta de forma a tornar persistentes as alterações. A gestão da memória principal desempenha assim um papel crítico na concretização das propriedades de durabilidade e atomicidade transacionais.

2.1.2 *Controlo de concorrência*

Duas transações são concorrentes se durante a sua execução se sobrepõem no tempo. Quando um conjunto de transações executam concorrentemente, as suas operações podem ser intercaladas.

Uma história representa a ordem pela qual as operações das transações são executadas entre si. Duas operações estão em conflito se forem executadas por diferentes transações no mesmo item de dados e pelo menos uma das operações é de escrita [6].

Permitir a execução de transações concorrentes que manipulam os mesmos dados de uma base de dados, pode levar à ocorrência de anomalias nos dados [4]. As anomalias possíveis são *Lost Update*, *Dirty Read*, *Nonrepeatable Read*, *Phanthom Read* e *Write Skew*.

LOST UPDATE Ocorre quando duas transações T_1 e T_2 leem o mesmo item de dados e ambas modificam esse item de dados independentemente uma da outra.

DIRTY READ Ocorre quando uma transação T_1 lê um item de dados modificado por outra transação T_2 que ainda esteja no estado ativa. No caso de a transação T_2 ser abortada, o item lido por T_1 nunca foi confirmado, ou seja nunca existiu.

NONREPEATABLE READ Ocorre quando um transação lê um item de dados e outra transação T_2 modifica ou elimina o mesmo item de dados e faz *commit*. Se a transação T_1 voltar a ler o mesmo tuplo de dados, obtém uma valor diferente ou descobre que este foi eliminado.

PHANTHOM READ Ocorre quando uma transação T_1 lê um conjunto de dados que satisfazem uma determinada condição de pesquisa. Se uma transação T_2 escrever itens de dados que afetam a condição de pesquisa e T_1 repetir a leitura com a mesma condição, obtém um conjunto de dados diferentes da primeira leitura.

WRITE SKEW Ocorre quando duas transações T_1 e T_2 leem o mesmo conjunto de itens de dados (e.g. T_1 lê x e T_2 lê y), mas modificam itens de dados diferentes desse conjunto (T_1 modifica x e T_2 modifica y). Se houver uma restrição entre x e y (e.g. $x + y > 0$), então esta restrição poderá ter sido violada.

O controlo de concorrência tem como objetivo garantir que as transações são executadas corretamente e sem provocar a ocorrência de anomalias nos dados, ou seja, garantir a propriedade de isolamento.

Existem quatro níveis de isolamento definidos no standard ANSI SQL caraterizados de acordo com a possibilidade de ocorrência de anomalias. Os níveis de isolamento são *read uncommitted*, *read committed*, *repeatable read* e *serializable*.

READ UNCOMMITTED Garante que as escritas de uma transação são isoladas das escritas de outras transações. Permite a ocorrência das anomalias referidas.

READ COMMITTED Garante que uma transação apenas pode ler itens de dados escritos por transações confirmadas. Impede a ocorrência da anomalia *Dirty Read*.

REPEATABLE READ Garante que uma transação apenas pode ler itens de dados escritos por transações confirmadas e que outras transações não possam ler os dados escritos pela transação atual até que esta termine. Permite apenas a ocorrência da anomalia *Phanthom Read*.

SERIALIZABLE Garante que uma transação apenas pode ler itens de dados escritos por transações que tenham sido confirmadas, que outras transações não possam ler os dados escritos pela transação atual até que esta termine e que outras transações possam escrever dados que afetem condições de pesquisa da transação atual até que esta termine. Impede assim a ocorrência de qualquer anomalia.

Existem duas abordagens para o controlo de concorrência, a abordagem conservadora e a abordagem otimista. Estas diferem em dois aspetos, quando é feita a deteção de conflitos e como são resolvidos os conflitos.

Na abordagem conservadora a deteção de conflitos é feita na submissão das transações implicando que transações que possam entrar em conflito, sejam impedidas de executar concorrentemente.

A abordagem alternativa é conhecida como controlo de concorrência otimista (OCC) [19]. Nesta abordagem, uma transação nunca é bloqueada ao executar as suas operações, mas poderá ser abortada depois de submetido o pedido de *commit*. A execução de transações assume otimismo quanto à ocorrência de conflitos, ou seja, que não devem ocorrer conflitos durante a execução da transação. A deteção e resolução de conflitos é realizada em tempo de *commit*. Quando uma transação é abortada, esta poderá ser reexecutada até que não ocorra nenhum conflito na sua execução.

As garantias de isolamento podem ser alcançadas através de diferentes técnicas. Técnicas de exclusão mutua, nomeadamente *locks*, são técnicas tradicionais que permitem às transações o acesso exclusivo aos itens de dados a si relacionados.

Locks são mecanismos que permitem às transações adquirirem e negarem o acesso a outras transações a itens de dados ao associar cada item dos dados com um *lock*. São utilizados *locks* de leitura (partilhados) para os dados que vão ser lidos e *locks* de escrita (exclusivos) para os dados que vão ser escritos. Dois *locks* de diferentes transações no mesmo item estão em conflito se pelo menos um deles for de escrita.

Os diferentes níveis de isolamento utilizam diferentes comportamentos para a aquisição de *locks* de leitura e escrita por transações concorrentes, com o objetivo de evitar diferentes anomalias.

É normalmente utilizado o protocolo *two-phase locking* (2PL) [6] para controlo da aquisição de *locks*. O protocolo ocorre em duas fases, na *growing phase* a transação obtém os *locks* para os itens de dados a que pretende aceder e de seguida executa as operações, na *shrinking phase* a transação liberta os *locks* obtidos.

Embora existam protocolos que permitam resolver o problema de *deadlock*, estes não o permitem resolver e proporcionar alta concorrência no sistema [19]. Protocolos de *locking* mais restritos como S2PL e SS2PL apresentam uma performance baixa com a execução de muitas transações [9].

Protocolos de *locking* seguem uma abordagem conservadora e utilizam o cancelamento de transações e *locks* para resolver os conflitos.

Na presença de poucos conflitos, OCC garante um melhor desempenho que protocolos de *locking*. Quando o conflito de transações é muito provável, a probabilidade de insucesso das transações aumenta. Consequentemente, uma elevada taxa de insucesso pode originar o problema de *starvation* devido às execuções repetidas das transações [19].

2.1.3 *Controlo de concorrência multiversão*

O controlo de concorrência multiversão (MVCC) [5] é um método que mantém uma lista de versões para cada item de dados. No MVCC cada escrita no item de dados x produz uma nova versão de x, em vez que atualizar o valor. É mantido assim uma lista de versões de x, que são a história dos valores que foram atribuídos a x. O método é responsável por determinar como as operações de leitura selecionam a versão correta dos dados para ler.

Uma vez que cada escrita num item de dados é gravada com uma versão diferente e os valores antigos não são alterados, o método não impede que outras transações façam operações de leitura a um item de dados que foi reescrito, permitindo assim um aumento de concorrência.

A desvantagem do método é o custo de armazenamento de múltiplas versões dos dados e a necessidade de proceder à remoção de versões obsoletas.

2.1.4 *Snapshot Isolation*

Snapshot Isolation (SI) é um protocolo de controlo de concorrência otimista [4] que assume uma base de dados MVCC e que permite às transações terem diferentes vistas do estado da base de dados, em que cada uma destas vistas corresponde a um instantâneo (*snapshot*) da base se dados. Uma transação é definida por uma versão inicial (*Ts*) atribuída no inicio da transação, e por uma versão final (*Tc*) atribuída no momento em que o pedido de *commit* é autorizado. As operações de escrita da transação são refletidas no seu próprio instantâneo, para que possam ser lidas caso a própria transação volte a ler os dados novamente. Escritas feitas por outras transações iniciadas depois do *Ts* são invisíveis à transação. O pedido de *commit* é bem sucedido apenas se o *writeset* da transação não foi modificado por outras transações a partir do momento em que o instantâneo foi tirado. Isto é chamado de conflito *write-write* e acontece quando duas operações de sobrepõem no tempo e têm interação nos seus *writesets*. Para transações que estejam em conflito, vence a transação que efetuou primeiro o *commit*, ou seja, vence a transação com o menor *Tc*, pelo que outras transações em conflito são abortadas. Transações que apenas tenham operações de leitura, nunca estarão em conflito com outras transações, uma vez que apenas precisam de ler a versão correta dos dados, o que permite que nunca sejam bloqueadas e sejam sempre confirmadas na deteção de conflitos.

SI evita as anomalias descritas no standard ANSI SQL [4], no entanto apresenta a anomalia *write skew*. Para muitas aplicações o *write skew* não é um problema para a integridade dos dados, mas pode sê-lo para aplicações que definam invariantes externos sobre a alteração concorrente de um conjunto de itens de dados, se cada transação apenas escrever um subconjunto destes.

2.1.5 *Serializable Snapshot Isolation*

Por definição o nível de isolamento *Serializability* garante que não ocorrem anomalias. O nível de isolamento Serializable Snapshot Isolation (SSI) que fornece as garantias de *Serializability* utilizando SI ao detetar potenciais anomalias em tempo de execução e procedendo ao *abort* das transações se necessário [8]. Na primeira implementação baseada em SSI [25] é utilizado um gestor de *locks* otimizado para seguir as dependências de leituras do SSI, tornando-o mais simples que os gestores de *locks* clássicos.

2.1.6 *Mecanismos de recuperação*

Um sistema transacional precisa de resolver o dilema entre a atomicidade e a durabilidade, garantindo que os efeitos das transações confirmadas, e apenas estes, persistem na sua

totalidade. Em caso de falha e reinício do sistema, admite-se que é efetuado um processo de recuperação antes do sistema retomar a operação normal, que corrige eventuais violações da atomicidade usando informação auxiliar sobre as transações em curso no momento da falha, guardadas num registo sequencial (*log* ou *journal*).

Este processo de recuperação pode ser concretizado de duas formas diferentes, com diferente impacto no desempenho do sistema. A primeira alternativa, *undo*, consiste em remover os efeitos de transações incompletas. Neste caso, o registo guarda os valores anteriores à modificação e os blocos modificados pela transação em curso podem ser imediatamente escritos na base de dados. Tem a vantagem de não obrigar a manter as modificações não confirmadas em memória, pois podem ser escritas imediatamente, mas obriga a fazer todas as alterações antes de confirmar a transação.

A segunda alternativa, *redo*, consiste em refazer os efeitos de transações incompletas. Neste caso, a transação em curso não pode escrever diretamente na base dados guardando os novos valores no registo sequencial. Tem a vantagem de não obrigar a escrever quaisquer modificações na base de dados, acelerando assim a confirmação da transação, mas obriga a manter todos os blocos alterados em memória até à confirmação, o que limita a dimensão da transação.

A maioria dos sistemas transacionais usa por isso um mecanismo híbrido, que executa ambos os protocolos em simultâneo, escrevendo tanto os valores passados como os futuros no registo sequencial. Isto permite que transações de grande dimensão possam ir escrevendo na base de dados mas reduz o tempo de espera para a confirmação, que exige apenas que seja completada a escrita para o registo sequencial [22].

2.2 BASES DE DADOS NOSQL DISTRIBUÍDAS

Uma base de dados pode escalar com o aumento das operações de leitura, operações de escrita e do seu tamanho. Existem duas técnicas para escalar uma base de dados: replicação e *sharding*.

Na replicação os dados são armazenados em mais do que um nó do sistema. Esta técnica permite aumentar a performance das operações de leitura ao distribuir as operações pelos vários nós. Uma vez que os dados são replicados por vários nós, em caso de falha de um nó, deverá existir pelo menos um nó no sistema capaz de o substituir. A replicação dos dados pode ser feita também por diferentes zonas geográficas, garantindo a disponibilidade do sistema em caso de falha de uma zona e reduzir a latência para clientes mais próximos de cada zona. Por outro lado as operações de escrita têm de ser replicadas por cada nó do sistema. Existem duas alternativas para o fazer, na primeira uma operação de escrita é feita de forma síncrona em todas as réplicas, na segunda a operação é feita de forma síncrona numa ou num número limitado de réplicas e de forma assíncrona nas restantes.

O *sharding* consiste em dividir o armazenamento dos dados por vários *shards*, que podem ser distribuídos por vários nós do sistema. A divisão dos dados pode ser feita, por exemplo, por uma função de *hash* que aplica a chave primária a um item de dados para determinar o respetivo *shard*. Desta forma novos nós podem ser adicionados ao sistema para aumentar a capacidade de armazenamento e a performance das operações de leitura e escrita. A desvantagem da técnica é que a divisão dos dados torna algumas operações típicas em bases de dados complexas e ineficientes.

Quantos mais nós são adicionados ao sistema, ao usar a técnica de *sharding*, maior é a probabilidade de que um nó do sistema possa falhar. Desta forma a técnica de *sharding* é frequentemente combinada com a replicação, tornando o sistema tolerante a faltas.

2.2.1 *Teorema CAP e BASE*

A replicação e *sharding* dos dados garante a disponibilidade do sistema, mas ao garantir a coerência dos dados implica que estes fiquem temporariamente indisponíveis. Por outro lado, permitir a incoerência dos dados garante a disponibilidade destes. Este *trade-off* é uma consequência central do teorema CAP [7], segundo o qual um sistema distribuído apenas é capaz de garantir simultaneamente duas das seguintes propriedades: Coerência(C), Disponibilidade(A) e Tolerância à Partição(P).

COERÊNCIA O sistema é coerente se após a atualização do seu estado, qualquer leitura subsequente obtém o valor de dados mais recente ou um erro. Isto implica que todos os nós do sistema vêm os mesmos dados, ao mesmo tempo.

DISPONIBILIDADE O sistema deve estar disponível durante um determinado período de tempo e qualquer pedido deve receber uma resposta independentemente do estado de qualquer nó do sistema.

TOLERÂNCIA À PARTIÇÃO O sistema deve continuar a funcionar corretamente, mesmo perante a perda de um número arbitrário de mensagens na rede ou perante a falha parcial do sistema.

Nenhum sistema distribuído é livre de falhas na rede, pelo que numa base de dados distribuída a propriedade de tolerância à partição deve ser garantida. Isto implica que numa base de dados distribuída, das propriedade de coerência e disponibilidade, apenas uma pode ser escolhida.

Em bases de dados distribuídas podemos ter coerência forte ou coerência fraca. Sistemas de coerência forte implementam as propriedades ACID, garantem que após uma operação de escrita, qualquer operação de leitura subsequente devolve um valor atualizado. Sistemas de coerência forte garantem as propriedades de Coerência e de Tolerância à Partição.

Por outro lado, temos sistemas de coerência fraca que não garantem que após uma operação de escrita será devolvido o valor atualizado para uma operação de leitura subsequente. Deste nível de coerência advém a coerência eventual BASE (Basically Available, Eventual, Consistency) que garante que se durante um determinado tempo não existirem operações, todas as operações serão propagadas por todos os nós do sistema, ficando assim o sistema coerente. Sistemas de coerência fraca garantem as propriedades de Disponibilidade e de Tolerância à Partição.

2.2.2 *Sistemas de bases de dados NoSQL*

O termo NoSQL foi originalmente usado em 1998 por Carlo Strozzi como o nome de uma base de dados que não utiliza uma interface SQL [2]. O termo foi reintroduzido em 2009 como acrónimo para "Not Only SQL", que define um sistema de gestão de grandes volumes de dados e que não usa unicamente SQL.

As bases de dados NoSQL são baseadas em BASE e são desenvolvidas com o objetivo de serem flexíveis, altamente disponíveis e horizontalmente escaláveis ao contrário dos RDBMS que possuem uma estrutura rígida e não escalam horizontalmente. Foram desenvolvidas várias propostas de bases de dados NoSQL como o Bigtable [10], DynamoDB [13], Cassandra [20] e o HBase [1].

Os sistemas de bases de dados NoSQL são assim caraterizados por serem livres de um esquema de dados. Os dados são organizados num nível lógico e acedidos apenas pela sua chave. Não é assim possível consultar os dados através de operações SQL, como *join* e *group by*, mas permite o fácil armazenamento e recuperação de dados independentemente da sua estrutura, assim como alterações nas aplicações cliente não implicam uma mudança no esquema de dados.

Os dados podem assim ser distribuídos verticalmente, em que as partes de um registo são distribuídos por um ou mais nós e horizontalmente em que os registos são distribuídos por vários nós (*sharding*). Isto permite que as bases de dados NoSQL escalem horizontalmente uma vez que um conjunto de servidores é responsável pela gestão e processamento dos dados.

Diferentes sistemas propostos, apresentam diferentes otimizações para coerência dos dados, estratégias de replicação, tipos de dados e esquema de dados. Os sistemas de bases de dados NoSQL podem ser divididos em 3 categorias.

KEY-VALUE STORES Todos os dados são armazenados e representados por um par chave-valor por registo. Esta estrutura é também conhecida como "hash-table" e o acesso aos dados é feito através da sua chave que permite aceder aos dados do registo, representado como valor.

DOCUMENT-STORE São desenhados para a gestão de dados armazenados em documentos que utilizam diferentes formatos standard, como XML e JSON. Fornecem um mecanismo de pesquisa simples.

GRAPH DATABASE Desenhados para armazenar dados que podem ser representados como um grafo com elementos interligados, como redes sociais ou rotas de transporte. Estruturalmente os nós são representados por entidades, as arestas representam as relações entre as entidades e as entidades e relações podem ter atributos.

2.2.3 *HBase*

O HBase é uma base de dados NoSQL distribuída e altamente escalável. É um implementação *open-source* do Bigtable que corre sobre HDFS [26] fornecendo as capacidades do Hadoop.

Uma tabela é constituída por linhas que são identificadas pela sua *row key*. Os dados de uma linha são agrupados em famílias de colunas (*column families*). Os dados de uma família são endereçados por um qualificador (*column qualifier*).

A combinação de uma *rowkey*, família e qualificador, identifica unicamente uma célula de dados. As células suportam multi-versão através da utilização de versões.

As principais propriedades do esquema de dados do HBase são as seguintes [18]:

- Índices das tabelas são baseados apenas na sua *row key*
- As tabelas são armazenadas e ordenadas com base na *row key*.
- Não existem tipos de dados e os dados são armazenados em conjuntos de *bytes*
- Linhas têm coerência forte dos dados, contudo não existem as mesmas garantias entre linhas ou tabelas, ou seja, não há transações.
- Famílias de colunas devem ser definidas na fase de criação da tabela. Não são facilmente alteradas uma vez que têm impacto na distribuição física dos dados.
- Qualificadores são dinâmicos e podem ser adicionados em tempo de execução.

Arquitetura

O HBase reside numa arquitetura *master-slave*. As tabelas são repartidas horizontalmente em Regiões. Cada Região é persistida no sistema de ficheiros HDFS e são geridas por instâncias denominadas por *DataNodes*. As Regiões são distribuídas por diversos servidores denominados por Servidores de Região. Enquanto que cada *DataNode* é responsável pela leitura e escrita dos dados, o Servidor de Região é responsável pela comunicação com o cliente para o acesso aos dados.

A coordenação dos Servidores de Região e as operações de administração (criar, apagar e atualizar tabelas) é feita pelo Master. O *ZooKeeper* é utilizado para coordenar o estado da informação partilhada.

2.3 TRANSAÇÕES EM NOSQL

2.3.1 *Arquitetura de NoSQL transacional*

As propriedades de escalabilidade e alta disponibilidade das bases de dados NoSQL são conseguidas com o relaxamento das garantias transacionais, ou seja, não fornecem transações com propriedades ACID, oferecendo uma coerência fraca à custa da coerência eventual dos dados.

No entanto, com o aumento da popularidade destes sistemas, têm sido apresentadas várias propostas que permitem o suporte transacional sobre estes sistemas, com o objetivo de beneficiar da sua escalabilidade e garantir a concorrência e atomicidade fornecida por sistemas de gestão de transações.

2.3.2 *Alternativa Undo*

Neste mecanismo de recuperação, para iniciar a transação o cliente envia um pedido de *begin* ao gestor transacional. Depois de iniciada a transação o cliente escreve diretamente na base de dados, e como não conhece averão final (T_c), escreve o identificador da transação como versão. Para ter isolamento é preciso escrever meta-informação extra. Outros clientes filtram pela meta-informação ao ler, para determinar se os itens lidos foram confirmados e são visíveis no seu instantâneo. Depois da confirmação da transação, o cliente deverá atualizar a meta-informação com o T_c, marcado os itens escritos como confirmados.

A versão inicial (T_s) é utilizada para ler do instantâneo da base de dados. Entre a confirmação da transação e a atualização da meta-informação, os itens podem ser lidos por outras transações, pelo que em caso de leitura de um item por confirmar a transação deverá consultar o gestor transacional para determinar se o item é visível no seu instantâneo. As escritas efetuadas pelo cliente devem ser adicionadas ao conjunto de escritas da transação para deteção de conflitos.

Após terminar de executar as suas operações o cliente envia um pedido de confirmação ao gestor transacional. Este atribui um T_c à transação, faz deteção de conflitos e responde ao cliente. Em caso de a transação ser confirmada, o cliente deve atualizar a meta-informação dos itens escritos com o T_c da transação, ou, caso contrário, removê-los. A transação é assim considerada completa.

2.3.3 *Alternativa Redo*

Neste mecanismo de recuperação, para iniciar uma transação, o cliente envia um pedido de *begin* ao gestor transacional. Em vez de escrever diretamente para a base de dados, o cliente

escreve para uma *cache* local, já com o T_c como versão. O T_s é utilizado para ler itens da base de dados. Contudo para ler as próprias escritas (RYOW), deverá ler também da *cache* local. As escritas efetuadas pelo cliente devem ser adicionadas ao conjunto de escritas da transação para detecção de conflitos. Para garantir atomicidade, antes de enviar o pedido de confirmação para o gestor transacional, as escritas em *cache* devem ser persistidas num registo sequencial.

Para garantir isolamento seria necessário bloquear os pedidos de *begin* no gestor transacional, até que as transações confirmadas sejam aplicadas. Existem, contudo critérios alternativos de isolamento que permitiriam simplesmente "congelar" o T_s a ser devolvido, não o avançando até não haver aplicações pendentes [14].

Em caso de a transação ser confirmada, o cliente aplica as alterações na base de dados e envia e notifica o gestor transacional, para que este possa avançar o T_s. Caso contrário, basta limpar a *cache*. A transação é assim considerada completa.

2.3.4 *Sistemas Transacionais*

Uma das primeiras propostas foi implementar SI sobre BigTable, combinando trincos com a confirmação de transações em duas fases (2PC) e um mecanismo de recuperação *undo* [24] efectuado pelos clientes. Permite a existência de transações multi-linha, ao adicionar duas colunas extra para cada coluna da base de dados (*lock* e *write*). A coluna *write* é usada para armazenar as versões finais de cada transação. A coluna *lock* é usada para simplificar a detecção de conflitos entre escritas ao permitir que o protocolo 2PC mantenha a escrita numa coluna bloqueada, impedindo que outras transações possam progredir até que o trinco seja libertado. Além disso, a manutenção da meta-informação das transações nas tabelas de dados coloca carga adicional aos servidores de dados. O código-fonte da implementação não está disponível.

Outra proposta implementa SI sobre HBase utilizando filas implementadas usando tabelas, de forma a ordenar a certificação das transações [27]. Tal como o proposto em [24], não é utilizado um gestor de transações, sendo o processo de certificação feito por cada cliente através da meta-informação das transações mantida no HBase. O mecanismo de recuperação *undo* também é efetuado pelos clientes. Apresenta algumas desvantagens na medida em que o mecanismo de implementação das filas requer várias leituras completas das tabelas de meta-informação para a detecção de conflitos. O código fonte também não está disponível.

Posteriormente, surgiram propostas de implementação de SI sobre bases de dados NoSQL utilizando gestores de transações dedicados.

Com base no proposto em [24], foi proposta uma implementação que utiliza um gestor de transações centralizado que permite evitar a utilização de trincos [15]. Este gestor de transações, *Transaction Oracle* (TO), é responsável por gerar números de versão, bem como

pela gestão e certificação das transações, incluindo um mecanismo de recuperação *undo*. De forma a evitar a utilização de trincos, é sincronizada uma pequena quantidade de meta-informação da transação com o cliente, que é responsável por comunicar com o TO para fazer os pedidos de *begin* e *commit*.

Outra abordagem é a implementação de um mecanismo de recuperação *redo* sobre uma base de dados NoSQL sem suporte multi-versão, como Apache Cassandra[1] [11]. Um gestor transacional é utilizado para manter o contexto transacional das transações em estado ativo. As operações de leitura e escrita são feitos pelo cliente através do gestor transacional. Não tendo suporte multi-versão na base de dados, é utilizado um repositório de versões não persistente (RVPN) para guardar as versões antigas dos itens de dados, sendo a versão de dados mais recente de cada item guardada na base de dados. O gestor transacional utiliza o TO do proposto em [15] para obter números de versão para as transações e fazer a sua certificação. As escritas apenas são feitas na base de dados depois da confirmação da transação.

Também é possível implementar um mecanismo de recuperação híbrido, utilizando trincos [21]. Neste caso, o gestor transacional atribui um número de versão à transação, faz cada alteração, e escreve os valores anteriores e atuais do item num registo sequencial, notificando o cliente de cada alteração bem sucedida. Depois de todas as operações terem sido processadas, o gestor transacional marca a transação como confirmada no registo sequencial e notifica o cliente. O código-fonte da implementação não está disponível.

2.3.5 *Discussão*

A forma de atribuir os números de versão às transações é diferente nos mecanismos *undo* e *redo*. No *undo* o T_c só é conhecido depois de a transação ser confirmada, logo as escritas que sejam feitas antes do fim da transação têm que ser anotadas com meta-informação adicional e filtradas explicitamente durante a leitura. Já no *redo*, como o T_c já é conhecido desde o inicio da transação, as escritas já são feitas com a versão final e apenas depois de a transação ser confirmada, possibilitando assim leituras por prefixo. No entanto, ao contrário do que acontece em bases de dados tradicionais, a memória não é partilhada entre todos os clientes pelo que as alterações efetuadas não ficam disponíveis imediatamente. Ou seja, após confirmar uma transação t com T_c, não é possível começar imediatamente uma transação t com um $T'_s = T_c$, dado que o as escritas poderão ainda não estar aplicadas na base de dados.

Embora as propostas apresentadas utilizem técnicas bem conhecidas de bases de dados, a sua aplicação está enquadrada em pressupostos diferentes, não sendo claro que os compromissos tradicionais de mantenham. A maioria destas propostas são de implementação

1 http://cassandra.apache.org

fechada e não existe um estudo que sistematize os compromissos relacionados com a escolha do mecanismo de recuperação e qual o seu impacto na arquitetura do sistema.

3

IMPLEMENTAÇÃO

Para estudar os comportamentos dos mecanismos de recuperação *undo* e *redo* interessa-nos implementar, avaliar e comparar ambas as alternativas. Neste capitulo apresentamos a arquitetura e a implementação de uma camada de *middleware* das duas alternativas.

3.1 ARQUITETURA

Neste trabalho assumimos como referência o sistema Apache HBase, semelhante à proposta original do *Bigtable*. A interface permite ler células, linhas, ou sequências de linhas, ordenadas pela sua chave. A leitura pode obter a última versão de cada célula ou ler células com número de versão inferior a um limite. Isto permite ler apenas um prefixo das operações de leituras efetuadas. O número de versão é indicado explicitamente ao escrever, sendo por omissão usado o tempo real.

Na concretização de transações assumimos a utilização do sistema NoSQL sem quaisquer modificações. Um cliente transacional utiliza uma biblioteca transacional que oferece um gestor transacional, para delimitar as transações, e uma versão da interface que permite fazer leituras e escritas no contexto de transações. A implementação de transações como uma camada cliente pode pois intercetar e modificar as operações de leitura e escrita, por exemplo, para acrescentar meta-informação adicional. Estas camadas em diferentes clientes precisam ainda de comunicar entre si para coordenação, proporcionando assim o isolamento entre transações.

Estas diferenças de arquitetura entre a camada transacional em NoSQL e a forma como as transações são concretizados no contexto de um SGBD tradicional têm consequências importantes para o desempenho dos mecanismos de recuperação. A principal decorre da interface cliente-servidor entre a camada que implementa transações e o armazenamento dos dados nos servidores, que restringe as operações possíveis e resulta num custo adicional em cada operação. Como consequência, torna-se imperativo tirar o maior partido da possibilidade de leitura de um prefixo através do número de versão.

A Figura 1 mostra a arquitetura com os componentes do sistema e como estes interagem entre si. Um Cliente Aplicacional faz uso de uma Biblioteca Transacional, que é responsável

por utilizar o Gestor Transacional para iniciar e terminar transações e o HBase como a base de dados onde serão efetuadas as operações de leitura e escrita

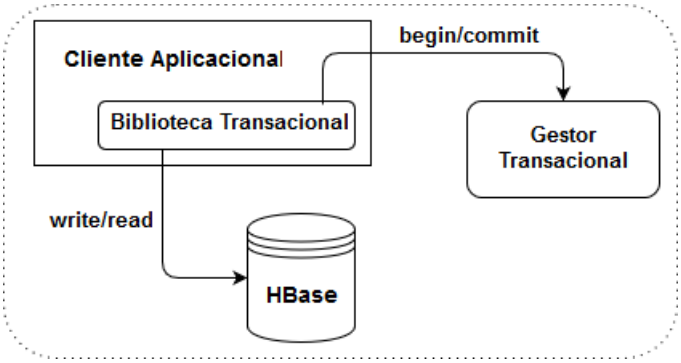


Figure 1: Arquitetura do sistema

3.1.1 *Gestor Transacional*

O Gestor Transacional (GT) é um componente centralizado responsável por oferecer um contexto transacional aos clientes e fazer a gestão das transações a decorrer no sistema.

Transação

Uma transação é definida um identificador T_{id}, pela versão inicial T_s usada para determinar que informação é visível no contexto da transação, por uma versão final T_c usada para marcar a versão da informação escrita pela transação e por um *writeset* transacional.

Writeset

O *writeset* transacional consiste no conjunto das operações de escrita feitas no contexto da transação. Na prática cada entrada no *writeset* identifica uma célula do HBase através de um *hash* ao nome da tabela, chave, *column family* e *column qualifier*. Isto permite ao GT fazer a deteção de conflitos entre as transações.

Deteção de conflitos

Para a deteção de conflitos o GT utiliza uma tabela de *hash* em memória, em que cada entrada na tabela é composta pelo *hash* de uma célula do HBase e a sua versão, que corresponde ao T_c da última transação que escreveu nessa célula.

O algoritmo 1 é usado em ambas as implementações para a deteção de conflitos e para uma dada transação é responsável por validar que, para todas as entradas no seu *writeset* (linha 6), o T_s da transação é maior que a versão da respetiva entrada da tabela (linha 8).

Se a transação for confirmada, todas as entradas no *writeset* da transação são atualizadas na tabela de *hash* com o T_c da respetiva transação (linha 12).

Algorithm 1 Deteção de conflitos

```
1:  var hashTable
2:  procedure ConflictDetect(Tx)
3:      Ts ← Tx.Ts
4:      Tc ← Tx.Tc
5:      writeset ← Tx.writeset
6:      for all cellId ∈ writeset do
7:          lastTc ← hashTable.getLastetTc(cellId)
8:          if lastTc > Ts then
9:              return ABORT
10:         end if
11:     end for
12:     hashTable.UpdateTcWrites(Tc, writeset)
13:     return COMMIT
14: end procedure
```

Interface Cliente

Cada uma das implementações oferece uma interface mínima ao cliente transacional composta pelos métodos de *begin* e *commit* que permitem iniciar e confirmar uma transação. Dependendo da implementação, poderão ser oferecidos outros métodos ao cliente transacional.

begin Invocado pelo cliente para iniciar uma transação. A partir daqui o cliente pode executar operações a partir do contexto transacional obtido.

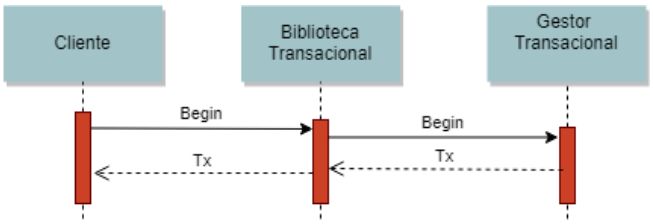
commit Invocado pelo cliente para fazer a confirmação da transação. O GT efetua a deteção de conflitos e devolve a confirmação para o cliente.

3.1.2 *Biblioteca Transacional*

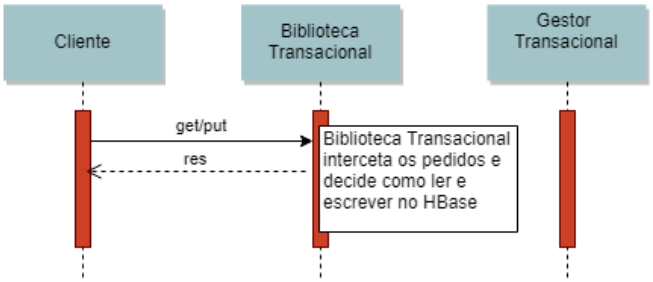
A Biblioteca Transacional (BT) oferece um GT ao Cliente aplicacional para que este possa obter um contexto transacional, sendo a BT responsável pela comunicação com o GT e pela gestão transacional no contexto do Cliente. O diagrama da Figura 2 ilustra a sequência de pedidos entre o Cliente, o GT e a BT no contexto de uma transação.

Para iniciar uma transação o Cliente utiliza a BT, que por sua vez é responsável por contactar o GT e devolver um contexto transacional para o Cliente (Figura 2a).

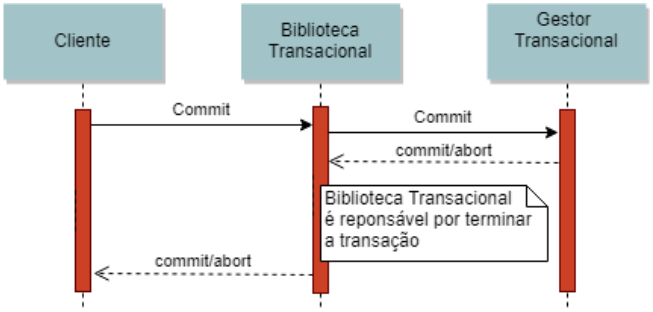
A partir do contexto transacional a interface do HBase é estendida, permitindo executar as operações oferecidas pelo HBase sobre o contexto da transação. A BT interceta



(a) Begin



(b) Read/Write



(c) Commit

Figure 2: Interação entre Cliente, Biblioteca Transacional e Gestor Transacional

as operações de leitura e escrita, modicando-as de forma a garantir o isolamento das transações (Figura 2b).

Para terminar a transação o Cliente volta a utilizar a BT, que contacta com o GT para confirmar a transação. A BT, ao receber a resposta do GT, termina a transação e devolve a resposta ao Cliente (Figura 2c).

3.2 PROTÓTIPOS

O protótipo consiste numa camada de *middleware* composta por uma implementação da biblioteca transacional e do gestor de transacional.

O *omid* é uma implementação da alternativa *undo* proposta em [15]. De forma a comparar experimentalmente as duas alternativas para o mecanismo de recuperação, usamos o *omid*

como concretização da alternativa *undo*. Este sistema está disponível como um projeto de código aberto e oferece suporte transacional multi-linha sobre o HBase.

De forma a ter uma concretização da alternativa *redo* tão semelhante quanto possível à anterior e tirando partido do código ser aberto, desenvolvemos uma versão modificada do *omid* que segue esta alternativa. Para o efeito, modificou-se a forma como as versões são atribuídas as transações e as escritas e leituras são feitas do HBase.

3.2.1 *Undo*

Gestor Transacional

Para a a atribuição das versões é usado um relógio lógico que devolve sempre a próxima marca temporal, garantido que a marca temporal devolvida é maior que a última devolvida. O T_s é atribuído no pedido de *begin* da transação (linha 3) e o T_c é atribuído no pedido de *commit*, antes da deteção de conflitos da transação (linha 7). O T_{id} da transação tem o mesmo valor do T_s.

Para garantir a atomicidade de uma transação, uma transação confirmada é persistida numa tabela de dados (CT) no HBase. Cada entrada na tabela consiste no mapeamento do T_{id} da transação para o seu T_c. Ou seja, para fazer o *commit* de uma transação o GT escreve o par (T_{id}, T_c) na CT (linha 9), garantindo assim a durabilidade da transação. Isto permite que em caso de leitura de um item de dados não confirmado, o cliente possa consular a CT para determinar se se trata de um item confirmado, não tendo assim que consultar o GT para determinar se o item lido pertence ao seu instantâneo.

Algorithm 2 Gestor Transacional - undo

```
1:  var Clock, CT
2:  procedure BEGIN
3:      Ts ← Clock.next()
4:      return new Transaction(Ts)
5:  end procedure
6:  procedure COMMIT(Tx)
7:      Tx.Tc ← Clock.next()
8:      if conflictDetect(Tx) == COMMIT then
9:          CT.insert(Tx.Tid, Tx.Tc)
10:         return COMMIT
11:     else
12:         return ABORT
13:     end if
14: end procedure
```

Biblioteca Transacional

A execução das transações decorre de forma otimista. Uma vez que o T_c só é conhecido após o *commit* da transação pelo GT, as escritas têm que ser anotadas com meta-informação adicional e filtradas explicitamente durante a leitura, se a transação ainda não tiver sido confirmada.

Para ter isolamento é preciso escrever meta-informação extra. Um campo de confirmação (cf) indica se um determinado item está confirmado, e se sim, qual o seu T_c. Este campo é inicialmente nulo, indicando que o item é uma escrita tentativa (não confirmada).

Na tabela 1 temos um exemplo de uma chave *key1* com duas versões, sendo a segunda uma escrita tentativa uma vez que que não tem um campo de confirmação *cq_cf* preenchido com a versão de *commit* da transação, nem a transação com o $T_id = 4$ se encontra na CT (tabela 2). Uma transação que faça a leitura da escrita tentativa, deverá consultar a CT para determinar se esta se trata de uma escrita confirmada e se pertence ao seu instantâneo.

Tabela de Dados

	ColumnFamily		
chave	**cq**	**cq_cf**	**versão**
key1	x	3	1
key1	y	nill	4

Table 1: Meta-informação numa tabela do HBase

Commit Table

T_id	T_c
1	3

Table 2: CT com o mapeamento (T_id, T_c) de uma transação confirmada, mas não completa

Para eliminar itens de dados, estes são escritos no HBase com a palavra reservada DELETE_TOMBSTONE como valor e devem ser filtrados nas operações de leitura.

As operações executadas pela biblioteca transacional são descritas no algoritmo 3.

BEGIN Cliente envia pedido de *begin* para o GT e obtêm um contexto transacional com o T_{id} e o T_s.

PUT Cliente escreve o item (chave, valor) no HBase com o T_{id} como versão (linha 7) e adiciona a escrita ao *writeset* da transação (linha 8).

DELETE Cliente escreve o item (chave, valor) no HBase com o T_{id} como versão e o valor com a palavra reservada DELETE_TOMBSTONE (linha 12). A escrita é adicionada ao *writeset* da transação (linha 13).

GET Cliente lê a partir do HBase itens com chave com a versão inferior ao T_s da transação (linha 16). Se um item lido tem o cf com uma versão menor que T_s, então essa leitura pertence ao instantâneo da transação e retornada para o cliente. Se for uma escrita tentativa (linha 17) a CT é consultada para determinar se o item já se encontra confirmado e se pertence ao instantâneo da transação (linha 22).

SCAN Idêntico à operação de *Get*, mas a procura é feita para um determinado range de valores.

COMMIT Cliente envia pedido de *commit* para o GT, que atribuí um T_c à transação, faz a deteção de conflitos e retorna a resposta para o cliente. No caso de confirmação da transação o cliente escreve o T_c no cf para todos os itens presentes no seu *writeset* (linha 33) e apaga a sua entrada da CT (linha 37), caso seja abortada, apaga as escritas feitas no HBase (linha 31). A transação é assim considerada completa.

3.2.2 *Redo*

Gestor de Transações

Nesta implementação as versões são atribuídas no início da transação, sendo por isso utilizados dois relógios para a sua atribuição: *ClockTS* para a atribuição do T_s e *ClockTC* para a atribuição do T_c (linhas 3 e 4). O T_{id} da transação tem o mesmo valor do T_c.

As transações têm uma ordem definida pelo seu T_{id} e que implica que uma transação só possa ser confirmada se todas as transações com T_{id} inferiores ao da transação já tiverem sido confirmadas ou abortadas.

Para garantir o isolamento da transação seria necessário bloquear o pedido de *begin* no GT até que uma transação confirmada seja terminada, para evitar isso o *ClockTS* é congelado, não avançando até que a execução da transação termine. Ou seja, no inicio da transação é devolvida a versão atual do *ClockTS* (linha 3) e o valor do *ClockTS* é apenas atualizado com o T_c da transação, após a confirmação da aplicação da transação (linha 23).

Assim que uma transação é iniciada pelo GT, esta é adicionada a uma fila (linha 6). Um processo no GT (linha 9) é responsável por olhar para a fila e tirar a próxima transação a ser confirmada. No entanto uma transação só pode ser confirmada quando o GT já tiver recebido um pedido de *commit* do cliente, pelo que é necessário esperar por esse pedido (linha 11). Assim o cliente faz o pedido de *commit* ao GT, o estado da transação é alterado como pronta para a deteção de conflitos (linha 19). Depois de feita a deteção de conflitos, a resposta é enviada para o cliente. Quando este terminar as suas operações, envia uma confirmação para o GT para que este possa avançar o *ClockTS* (linha 22).

Biblioteca Transacional

A execução das transações decorre de forma pessimista, ou seja, apenas são persistidas no HBase após a certificação da transação. Em vez de escrever para HBase, as escritas são assim feitas numa *cache* local, mas como T_c da transação já é conhecido, cada item de dados já tem o T_c como versão.

Algorithm 3 Biblioteca Transacional - undo

```
1: var GT, DS, CT
2: procedure BEGIN
3:     Tx ← GT.begin()
4: end procedure
5: procedure PUT(key, value)
6:     p ← new Put(key, value, Tx.Tid)
7:     DS.put(p)
8:     writeset.add(hash(p))
9: end procedure
10: procedure DELETE(key)
11:     p ← new Put(key, DELETE_TOMBSTONE, Tx.Tid)
12:     DS.put(p)
13:     writeset.add(hash(p))
14: end procedure
15: procedure GET(key)
16:     for rec ← DS.get(key, value, Tx.Ts) do
17:         if rec.cf != nil then
18:             if rec.cf < Tx.Ts then
19:                 return rec.value
20:             end if
21:         else
22:             return GetTentatiteValue(rec, key)
23:         end if
24:     end for
25: end procedure
26: procedure COMMIT
27:     GT.commit(Tx)
28:     for all key ∈ Tx.writeset do
29:         rec ← DS.get(key, Tx.Tid)
30:         if Tx.commit == true then
31:             delete rec
32:         else
33:             rec.cf ← Tx.Tc
34:         end if
35:     end for
36:     if Tx.commit == true then
37:         CT.delete(Tx.Tid)
38:     end if
39: end procedure
```

Algorithm 4 Gestor Transacional - redo

```
1: var ClockTS, ClockTC, Queue
2: procedure BEGIN
3:     Ts ← ClockTS.current()
4:     Tc ← ClockTC.next()
5:     Tx ← new Transaction(Ts, Tc)
6:     Queue.add(Tx)
7:     return Tx
8: end procedure
9: procedure SHEDULLER
10:    Tx ← Queue.next()
11:    Tx.waitForCommitRequest()
12:    if conflictDetect(Tx) == COMMIT then
13:        return COMMIT
14:    else
15:        return ABORT
16:    end if
17: end procedure
18: procedure COMMIT(Tx)
19:    Tx.State ← READY_TO_COMMIT
20: end procedure
21: procedure COMMITDONE(Tx)
22:    if ClockTS.Ts < Tx.Tc then
23:        ClockTS.Ts ← Tx.Tc
24:    end if
25: end procedure
```

O T_s é utilizado para determinar o instantâneo de leitura. Alem de ler da base de dados, para ler as próprias escritas (RYOW), deverá também ler da *cache* local.

Para eliminar itens de dados, estes são escritos com a palavra reservada DELETE_TOMBSTONE como valor e devem ser filtrados nas operações de leitura.

Em caso de certificação da transação, o cliente aplica as alterações no HBase, em caso de insucesso apenas tem de limpar a *cache* local.

As operações executadas pela biblioteca transacional são descritas no algoritmo 5.

BEGIN Cliente envia pedido de *begin* ao GT e obtém um contexto transacional com o T_{id}, T_s e o T_c.

PUT Cliente escreve o item (chave, valor) na sua *cache* local com o T_c como versão (linha 7) e adiciona a escrita a escrita ao *writeset* da transação (linha 8).

DELETE Cliente escreve o item (chave, valor) no HBase com o T_{id} como versão e o valor com a palavra reservada DELETE_TOMBSTONE (linha 12). A escrita é adicionada ao *writeset* da transação (linha 13).

GET Cliente lê a partir do HBase e da sua *cache* local itens com chave com a versão inferior ao T_s da transação (linha 16).

SCAN Idêntico à operação de *Get*, mas a procura é feita para um determinado range de valores.

COMMIT Cliente envia pedido de *commit* para o GT. No caso de ser confirmada, as escritas em *cache* são persistidas no HBase (linha 21) e um pedido de confirmação é enviado para o GT para que este possa avançar o *ClockTS* (linha 22), no caso de ser abortada as escritas em *cache* são descartadas (linha 24). A transação é assim dada como completa.

3.2.3 *Discussão*

Embora as alternativas *undo* e *redo* utilizem uma arquitetura semelhante e composta pelos mesmos componentes, o mecanismo de recuperação utilizado em cada uma implicam diferentes impactos no sistema pela forma como é feita a gestão transacional, recuperação, persistência e leitura dos dados.

A forma de atribuir números de versão é diferente nos dois métodos. Com *undo*, é utilizado o mesmo relógio lógico para a atribuição de T_s, no *begin* e T_c no pedido de confirmação. Com o *redo* são utilizados relógios diferentes para a atribuição T_s e T_c, e ambos são atribuídos no pedido de *begin* da transação, permitindo que o T_c seja logo conhecido.

Algorithm 5 Biblioteca Transacional - redo

```
1: var GT, DS, Cache
2: procedure BEGIN
3:     Tx ← GT.begin()
4: end procedure
5: procedure PUT(key, value)
6:     p ← new Put(key, value, Tx.Tc)
7:     Cache.put(p)
8:     writeset.add(hash(p))
9: end procedure
10: procedure DELETE(key)
11:     p ← new Put(key, DELETE_TOMBSTONE, Tx.Tid)
12:     DS.put(p)
13:     writeset.add(hash(p))
14: end procedure
15: procedure GET(key)
16:     return DS.get(key, value, Tx.Ts)
17: end procedure
18: procedure COMMIT
19:     GT.commit(Tx)
20:     if Tx.commit == true then
21:         Cache.flush()
22:         GT.CommiDone(Tx.Tid)
23:     end if
24:     Cache.clear()
25: end procedure
```

Devido a que, com *undo* as escritas são feitas diretamente no HBase, para garantir o isolamento é adicionada meta-informação aos itens de dados, que deve ser atualizada com o T_c após a confirmação da transação. No caso de a transação ser abortada, todas as escritas iniciais devem ser apagadas do HBase pela transação. Entre a primeira e segunda escrita, os itens escritos pela transação podem ser lidos por outras transações. Estas precisam de consultar o GT para saberem se esses itens pertencem ao seu instantâneo. No *redo* uma vez que o T_c já é conhecido a partir do inicio da transação, a escrita inicial, mesmo que feita inicialmente em *cache*, é já a escrita final uma vez que o item já tem como versão o T_c. Outras transações ao lerem os itens de dados, conseguem facilmente determinar se estes pertencem ao seu instantâneo uma vez que estes têm sempre o T_c como versão. No caso de a transação ser abortada, esta apenas precisa de limpar a sua *cache*, ou seja, não chega sequer a escrever no HBase.

Comparativamente, utilizando *undo*, é necessário fazer duas escritas por item no HBase no caso da transação ser confirmada, ou uma escrita e uma remoção no caso de não o ser, enquanto que na utilização do *redo* escreve-se uma só vez e apenas se a transação for confirmada. Embora avançar o T_s após confirmação da aplicação de transações pendentes, com *redo* impeça leituras de itens por confirmar por transações concorrentes, ao invés de ao usar-se *undo*, optando por "congelar" o T_s até à confirmação de aplicação da transação poderá provocar um aumento de conflitos, uma vez que o tempo de execução das transações aumenta, levando a um maior número de transações que não podem ser confirmadas.

A verificação de conflitos com *undo* é feita pela ordem de chegada do pedido de confirmação das transações, enquanto que, com *redo*, a verificação é feita pela ordem do T_id, o que implica que a verificação de uma transação só pode ser feita depois das transações ativas com T_id inferior. No entanto, se a transação for apenas de leitura, não precisa de esperar, já que não pode entrar em conflito com outras. Com *redo* o sistema depende dos clientes para avançar, ao invés de *undo*.

Tolerância a falhas

A alternativa *redo* apresenta dependência do cliente em duas fases, no pedido de confirmação da transação após o seu início (Figura 3) e na confirmação da aplicação da transação após a sua certificação (Figura 4). No caso de falha de um cliente, o sistema congela no tempo, não fazendo a certificação e transações com o T_id superiores ao da transação no primeiro caso, ou não avançando o $ClockTS$ no segundo caso.

O GT deverá ser responsável por detetar a falha de um cliente, no entanto, detetar se um cliente falhou ou se está apenas mais lento a responder traz desafios de implementação que poderão ter impacto no sistema. Uma vez que a decisão de qual a melhor estratégia a adotar se encontra fora do âmbito deste trabalho, assumimos uma versão otimista de que o cliente nunca falha.

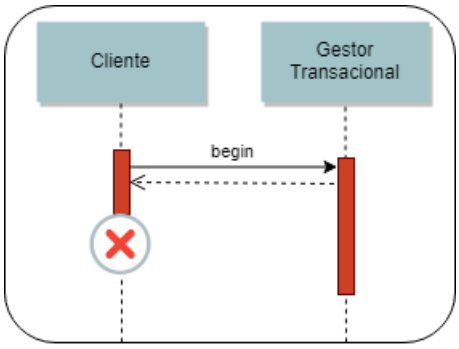


Figure 3: Falha do cliente antes do pedido de confirmação.

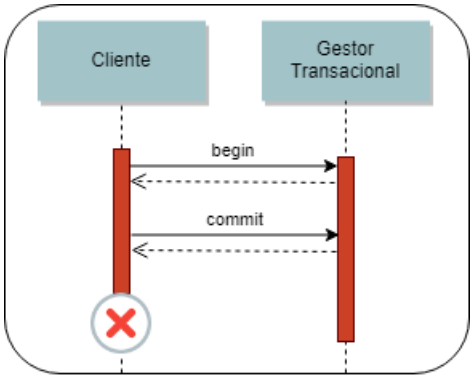


Figure 4: Falha do cliente antes da confirmação de aplicação da transação.

Transações demoradas

Uma vez que na alternativa *redo* as transações são certificadas pela ordem do T_id, se o sistema tiver transações muito mais demoradas que outras, poderá atrasar a certificação das transações mais rápidas e aumentar o seu tempo de execução. O AJITTS [23] consiste num mecanismo proposto para SGBD relacionais replicadas para minimizar a taxa de insucesso e maximizar o débito das transações, ao encontrar a altura ideal para executar cada transação submetida ao sistema. A sua aplicação poderá ser estudada neste sistema para resolver este problema.

4

AVALIAÇÃO EXPERIMENTAL

Neste capitulo com a utilização de um *benchmark* comparamos o desempenho das alternativas de acordo com as métricas de débito, latência, tempo de resposta e taxa de insucesso das transações. Comparamos também com a utilização do HBase sem garantias transacionais para mostrar o peso de adicionar estas garantias ao sistema.

4.1 YCSB

O Yahoo! Cloud Serving Benchmark (YCSB) [12] é um *benchmark* padrão da indústria para bases de dados chave-valor como o HBase. Foi desenvolvido com o objetivo de comparar diferentes sistemas de bases de dados NoSQL uma vez que o padrão de acesso aos dados é diferente dos sistemas tradicionais que foram projetados para bases de dados relacionais.

O *benchmark* é dividido em duas fases. A primeira, a fase de carregamento tem como objetivo inserir na base de dados um determinado número de itens de dados definido na carga de trabalho. A segunda fase, a fase de execução, consiste em executar o *benchmark*.

O YCSB começa por criar um número definido de clientes que vão executar um conjunto de operações na base de dados, de acordo com uma carga de trabalho definida. Existem quatro tipo de operações disponíveis: *Insert*, *Update*, *Read* e *Scan*.

A carga de trabalho permite várias opções de configuração, como as operações a executar, os itens que devem ser lidos ou escritos, o tamanho de cada campo e quantos itens deve ler a operação de *Scan*. Para além disso permite definir a distribuição de dados a ser utilizada.

A definição de uma carga de trabalho e uma distribuição de dados permite assim simular um cenário real.

4.2 CONDIÇÕES EXPERIMENTAIS

Uma vez que a implementação original do YCSB [3] não tem suporte transacional, modificámos o YCSB, adicionando transações que acedem e alteram múltiplas linhas.

Uma transação é definida na carga de trabalho da mesma forma que uma operação, mas na sua execução efetua um conjunto de operações de acordo com a transação definida.

Foram definidas sete tipos de transações de diferentes dimensões, das quais cinco são de escrita e duas de leitura. Para as transações definidas foram utilizadas diferentes cargas de trabalho que diferem na percentagem de leituras e escritas. Cada carga de trabalho foi testada com uma distribuição uniforme, em que as transações têm a mesma probabilidade de aceder a qualquer item de dados e uma distribuição *hotspot*, em que as transações têm mais probabilidade de aceder a um determinado conjunto de dados, provocando uma maior ocorrência de conflitos.

Para executar os testes foram utilizados quatro servidores (Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz, 8 GB RAM) com a mesma configuração e ligados numa rede *Gigabit*.

A Figura 5 demonstra a arquitetura utilizada para a avaliação. Um servidor com HBase instalado em modo *standalone*; um servidor com o GT instalado ; e os dois restantes servidores a executar como cliente, o *benchmark* YCSB.

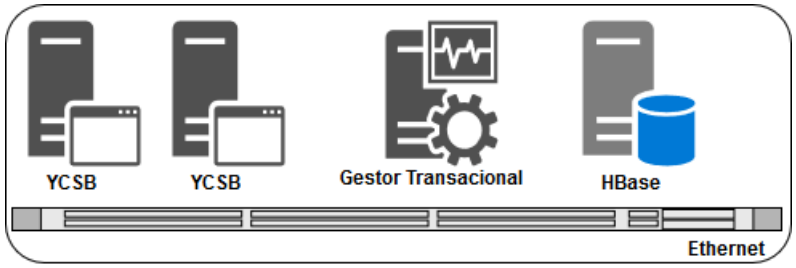


Figure 5: Arquitetura de avaliação experimental

Cada instância do YCSB correu 26 fios de execução, simulando, no total 52 clientes concorrentes. Cada teste executou no total 150000 transações.

Antes de cada teste, a base de dados foi carregada com 20000 linhas (*rows*), com 1 *column family* com 10 *column qualifier* cada. Cada linha tem aproximadamente 1KB de tamanho.

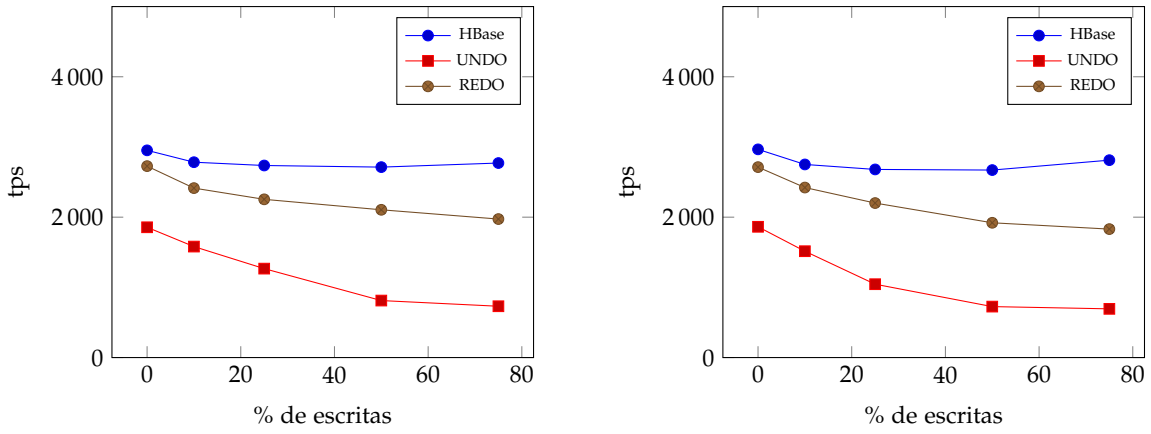
4.2.1 *Resultados*

Para além das implementações *undo* e *redo* descritas, foi também executado o *benchmark* sem contexto transacional (HBase) de forma a mostrar também o impacto de ter contextos transacionais sobre o HBase.

Os resultados apresentados nesta secção são calculados pela média de 3 execuções independentes.

Análise do débito

Como esperado o débito da utilização do HBase é superior ao das implementações do *undo* e *redo* (Figura 6). Uma vez que não existe contexto transacional cada transação apenas tem que escrever e ler diretamente no HBase. Contudo, não temos coerência dos dados.



(a) uniforme (b) hotspot

Figure 6: Débito variando a percentagem de transações de escrita e com diferentes distribuições de acesso a itens de dados.

A alternativa *redo* apresenta um débito superior em relação ao *undo*, sendo que a diferença aumenta à medida que a percentagem de escritas também aumenta, isto é, quanto maior a percentagem de escritas, melhor é o desempenho do *redo* em relação ao *undo* (Figura 6).

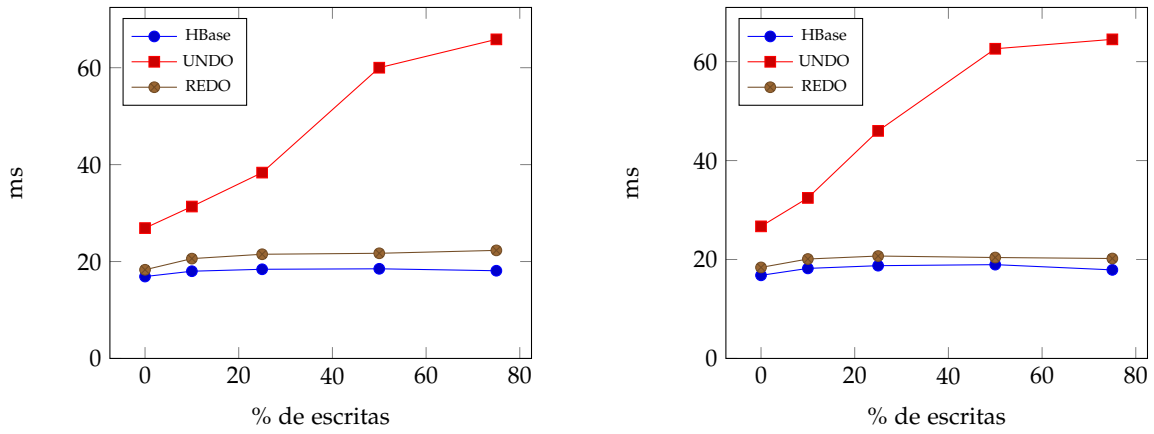
A diferença entre o *redo* e o *undo* quando a percentagem de escritas é próxima de o% é justificada pela necessidade do *undo* ler meta-informação dos itens de dados do HBase.

Ao comparar os gráficos da distribuição uniforme (Figura 6a) com a distribuição *hotspot* (Figura 6b) o débito desce com a distribuição *hotspot* uma vez que temos mais conflitos e consequentemente uma maior taxa de *aborts*. Ainda assim, na distribuição *hotspot* a diferença do débito mantém-se entre o *undo* e *redo*, mostrando que a alternativa *redo* tem um melhor desempenho que o *undo*, mesmo na presença de mais *aborts*.

Análise da latência

A latência das transações segue a tendência inversa ao débito. Com o aumento da percentagem de escritas a latência aumenta no *undo* e no *redo*, assim como com a utilização do HBase (Figura 7).

A latência do *redo* é muito próxima das utilização do HBase, apresentando uma ligeira subida com o aumento da percentagem de escritas. Esta aproximação acontece porque esta implementação tem de ler e escrever os mesmos itens dados que a utilização do HBase, tendo apenas como custo adicional a comunicação com o GT.



(a) uniforme (b) hotspot

Figure 7: Tempo de resposta variando a percentagem de transações de escrita e com diferentes distribuições do acesso a itens de dados.

Quanto ao *undo* a latência aumenta com a percentagem de escritas uma vez que nas operações de escrita tem de ser escrita a meta-informação adicional para cada item de dados. A diferença da latência quando a percentagem de escritas é próxima de 0% mais uma vez justificada pela necessidade do *undo* ler meta-informação dos itens de dados do HBase.

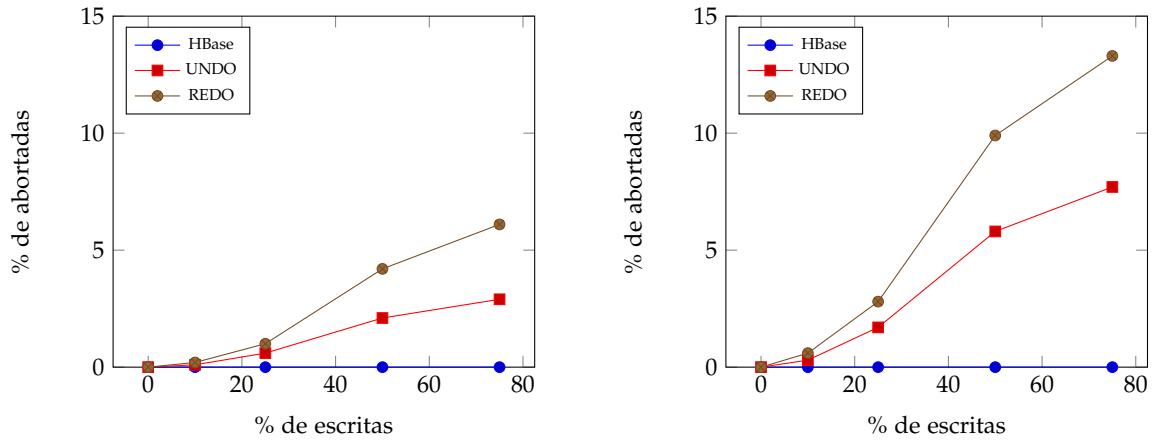
Ao comparar os gráficos da distribuição uniforme (Figura 7a) com a distribuição *hotspot* (Figura 7b), a latência apresenta valores idênticos, ainda que se note uma ligeira descida no *redo*. Essa descida é justificada pelo aumento de aborts. Uma vez que a deteção de conflitos é feita antes de se escrever no HBase, as transações em conflito abortam rapidamente, não prejudicando a execução de outras transações.

Análise de transações abortadas

Por fim, o número de transações abortadas é superior no *redo* em relação ao *undo*. Isto é justificado pelo maior débito do *redo*, uma vez que temos mais transações, a probabilidade de estas entrarem em conflito é maior (Figura 8).

Ao comparar os gráficos da distribuição uniforme (Figura 8a) com a distribuição *hotspot* (Figura 8b), a percentagem de transações abortadas aumenta com a distribuição *hotspot* nas implementações *undo* e *redo*, o que já era esperado uma vez que o objetivo da distribuição *hotspot* é aumentar o número de transações em conflito.

Na utilização do HBase não temos transações abortadas, pois não temos contexto transacional.



(a) uniforme (b) hotspot

Figure 8: Transações abortadas devido a conflitos variando a percentagem de transações de escrita e com diferentes distribuições do acesso a itens de dados.

5

CONCLUSÕES

Neste trabalho consideramos uma das principais decisões na concretização de suporte transacional multi-linha sobre o HBase: qual o impacto do mecanismo de recuperação escolhido para assegurar que as propriedades de atomicidade e durabilidade são respeitadas simultaneamente em caso de falha. Embora este seja um assunto estudado em profundidade em bases de dados tradicionais, a arquitetura de uma sistema transacional sobre uma base de dados NoSQL altera alguns pressupostos e torna necessária esta reavaliação. O objetivo era assim analisar e comprar o impacto dos mecanismos de recuperação *undo* e *redo*.

Para o efeito usámos uma implementação do método da alternativa *undo*, o omid, e desenvolvemos uma implementação para o método de recuperação *redo* baseada no omid, de forma a termos duas implementações o mais semelhante possíveis. Ambas as alternativas estendem a interface de uma base de dados NoSQL, o HBase, permitindo executar sobre este operações com um contexto transacional.

Para avaliarmos o desempenho das alternativas adaptamos o *benchmark* YCSB de forma a este suportar transações com contexto transacional e compararmos as duas alternativas.

Os resultados obtidos mostram que o mecanismo de recuperação *redo* apresenta um melhor desempenho que o *undo* à medida que temos mais escritas, sendo que a diferença de desempenho mantêm-se quando os sistemas têm de lidar com mais transações abordas. Mesmo com uma percentagem de escritas próxima de 0% o desempenho do *redo* consegue ser superior ao desempenho do *undo*.

Conclui-se então que, apesar do sistema mais difundido atualmente usar o método *undo*, vale a pena considerar o método *redo* de forma a lidar com aplicações mais exigentes em termos de escritas.

5.1 TRABALHO FUTURO

O objetivo deste trabalho consistia em avaliar e comprar o impacto das alternativas *undo* e *redo*. Embora à partida a alternativa *redo* apresente um melhor desempenho que o *undo*,

a forma como este deve lidar com a possibilidade de falha de clientes e o seu impacto no sistema deve ser estudada num futuro trabalho.

Seria ainda interessante testar ambas as alternativas com o HBase totalmente distribuído e com um *workload* superior ao utilizado.

BIBLIOGRAPHY

[1] Hbase. https://hbase.apache.org/. Accessed: 2016-12-22.

[2] Nosql. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. Accessed: 2016-12-22.

[3] ycsb. https://github.com/brianfrankcooper/YCSB. Accessed: 2017-06-20.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[5] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[7] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[8] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, Dec. 2009.

[9] M. J. Carey and M. Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases*, VLDB '84, pages 107–118, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[11] F. Coelho, F. Cruz, J. Pereira, R. Vilaça, and R. Oliveira. ph1: middleware transacional para nosql. 2013.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[14] S. Elnikety, F. Pedone, and W. Zwaenepoel. Generalized snapshot isolation and a prefix-consistent implementation. Technical report, 2004.

[15] D. Gómez-Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *IEEE 30th International Conference on Data Engineering*, pages 676–687, March 2014.

[16] J. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[17] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[18] A. Khurana. Introduction to hbase schema design. 2012.

[19] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[20] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[21] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, volume 11, pages 123–133, 2011.

[22] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[23] A. Nunes, R. Oliveira, and J. O. Pereira. Ajitts: Adaptive just-in-time transaction scheduling. In *Distributed Applications and Interoperable Systems DAIS*, page 57–70, Florence, Italy, June 2013. Springer, Springer.

[24] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[25] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, Aug. 2012.

[26] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[27] C. Zhang and H. D. Sterck. Hbasesi: Multi-row distributed transactions with global strong snapshot isolation on clouds. *Scalable Computing: Practice and Experience*, 12(2), 2011.